**Day 1 :**

We have 4 specific graphs that all do the same thing:   
        read a source file   
        aggregate the data   
        use the result to create a lookup file that is used by other graphs   
  
If we have a lot of these, it makes sense to create 1 generic graph and use parameters to describe the differences:   
        input file URL   
        input file record format   
        rollup key   
        rollup transform   
        output file URL   
        output file record format   
        additional optional components to do additional operations (normalizing the data)   
  
If we have a generic graph, we have three ways to get the parameter values for the graph:   
(1) Use a pset that some person configures in the GDE.   
(2) Use metaprogramming to \*compute\* some/all of the parameter values.   
(3) Create a template in Express>It and allow users to create configurations to fill in the parameter values.   
These three techniques can be combined — so you can use metaprogramming to compute some of the parameter values, use Express>It to enable some people to create psets for the remaining parameter values and/or use the GDE to enable other people to create psets for the remaining parameter values.   
  
  
When to use metaprogramming versus Express>It to get the value for a particular parameter?   
        — use metaprogramming when the parameter value can be specified as a variation on another parameter value.   
        examples: user selects the input record format, user selects 1 or more fields from the input record format, output record format = input record format minus those fields (we have a known method for computing the output record format)   
                                user selects the input record format, user selects 1 or more fields from the input record format, transform = max function applied to those fields (we have a known method for computing the transform rules)   
  
        — use Express>It when the parameter value can be literally anything — when there are no known algorithms/specifications/methods that always apply to create the record format or transform. In those cases, it’s better to create a really awesome template and let the user configure those in Express>It   
        you can use the record format editor and the ruleset editor controls to enable users to configure record formats, transforms, etc.   
  
By using metaprogramming, there is less work (fewer parameters) for the end user to configure. Metaprogramming saves time & effort for the end user when configuring psets for generic graphs.   
  
You can easily compute \*some\* of the parameters using metaprogramming. And whichever parameters don’t work well with metaprogramming could be configured using Express>It. You can combine these technologies as you see fit for your requirements.   
  
  
Typical design pattern for a graph that uses metaprogramming:   
        end user is asked to provide input specification (record format, file URL or table name) and some sort of information about how they want to proceed (list of fields to drop, which fields to aggregate, which key to use, etc.)   
        some or all of the remaining parameters are computed using metaprogramming functions (= this week’s class)   
  
  
=================================================   
  
How do parameters work in a graph?   
  
There are two types of parameters:   
        — “input parameters” : value for the parameter is provided by the end user (via pset, command-line, etc.) at run time   
        — “local parameters” : value for the parameter is computed by Co>Operating System at runtime.   
  
For local parameters, there are two values:   
        — “value” : that is assigned by the developer when the developer creates the graph   
        — “resolved value” : this is computed by the Co>Operating System at runtime by applying the Interpretation to its value. Resolved value is what is used by the components and transforms in the graph that reference the parameter. If a component has a reference to $ABC, then at runtime, the Co>Operating System substitutes the resolved value of ABC in place of that reference.   
  
  
  
Parameter Name                Value                                Interpretation        Resolved Value   
param1                                        xyz                                constant        xyz   
param2-bad                                $param1                        constant        $param1   
param2                                        $param1                        $-substitution        xyz   
param3                                        abc$param1                $-substitution        abcxyz   
param4-bad                                $param1\_abc        $-substitution        ERROR!   
        because it is ambiguous whether the parameter name is $param1 or whether it is $param1\_abc   
param4                                        ${param1}\_abc        ${}-substitution        xyz\_abc   
param5                                        $( id -n)                        shell        win\_rbuchheit   
param6                                        abc$(id - n)                shell        abcwin\_rbuchheit   
param7                                        $[ (date(“YYYYMMDD”))today())   PDL        20151026   
  
Interpretations:   
constant           
        resolved value = value   
        the value is read as-is and simply becomes the resolved value   
  
$-substitution   
        resolved value = value with any $-references replaced by the resolved value of that parameter   
  
${ }-substitution   
        same idea as $-substitution, except that parameter references are written as ${PARAM\_NAME}   
        why? in some cases, the $ reference can be ambiguous   
        ${ } clearly defines what is the parameter name and what isn’t   
  
shell   
        resolved value = value computed by the Korn shell by executing the command   
        anything inside $( ) is sent to the Korn shell & executed there.   
        the result of the execution is the value that is substitution in place of the $( )   
  
PDL (Parameter Definition Language)   
        PDL is a \*superset\* of all of the above Interpretations.   
        In the grid above, I could use PDL as the Interpretation for any of those parameters & I would get exactly the same result.   
        You can combine different types of syntax in the same parameter value and PDL will be able to resolve all of them.   
        So I could write this:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_today.dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
        PDL has an additional syntax that it understands, named “inline DML computation”   
        This syntax is written as $[ ]   
        Inside the $[ ], you can use Ab Initio DML syntax and functions to compute value.   
        You write a DML expression (= code that returns a single value), the resolved value of the   
                parameter is computed by the DML engine evaluating that expression.   
  
          
        In the latest version of the Co>Operating System & GDE, shell interpretation is no longer   
        an option. But you can still use $( ) syntax if you set the Interpretation to PDL — the code inside   
        $( ) will still be interpreted by the Korn shell. But there is no longer an option for shell   
        interpretation.   
        Why?   
        Shell interpretation is not recommended unless there is no other way to do it.   
        Using inline DML computation (typically the way that you would replace a shell interpreted   
        parameter) is \*faster\*, works better with dependency analysis, easier to use (you should   
        know how to write DML expressions, but you might not be so familiar with Korn shell   
        commands)   
        Why is inline DML computation faster than shell interpretation?   
                with inline DML computation, we’re using the Co>Operating System, computation is   
                        embedded in the graph (we don’t have to start anything extra)   
                with shell interpretation, we need to start a subshell from Korn shell, send the command,   
                        wait for the command to run, get the result, close the subshell for \*EACH\* shell   
                        interpreted parameter (=slower)   
        Why doesn’t dependency analysis work with shell interpretation?   
                security — there are commands that you might be allowed to run in dev that you can’t   
                        run (or don’t have permissions to run in prod)   
                filesystem versus EME — in the filesystem “rm $AI\_SERIAL/data” might be fine but if   
                        you do that in the EME, you could remove a lot of dataset objects that you wanted to   
                        keep. Running some commands in the EME might be very destructive in a way that   
                        you don’t expect (“rm -rf” would be disastrous in the EME!)   
  
  
        Why do we still have $-substitution around then? Why is that still an option?   
        There are a lot of cases where you simply want to write something like:   
                $AI\_SERIAL/myfile.dat — $-substitution is perfectly good for that   
                many of our components — by default — use $-substitution as the default parameter   
                interpretation for their parameters   
                backwards compatibility for our built-in components & their parameters   
  
                moving forward, it’s perfectly okay to use PDL for anything   
  
                  
============================   
  
A bit more on inline DML computation.   
  
Earlier, I wrote this:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_today.dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
  
You can combine inline DML computation with these other types of parameter references.   
        So I could write this as a parameter value:   
                $AI\_SERIAL/$(id -n)/${FILENAME}\_$[ (date(“YYYYMMDD”))today() ].dat   
        and it would be able to resolve correctly if the Interpretation was set to PDL.   
  
Inside the $[ ] (inline DML computation), there are some interesting syntax rules.   
For the examples below, assume all parameters are set to use PDL interpretation.   
  
Name                Value                                                                Resolved Value   
t\_date                $[ (date(“YYYYMMDD”))today() ]        20151026   
t\_year                $[ string\_prefix(t\_date, 4) ]                2015   
t\_month        $[ string\_substring(t\_date, 5, 2) ]        10   
t\_day                $[ string\_suffix(t\_date, 2) ]                26   
delim                ,                                                                        ,   
delim\_tx        string($delim)                                        string(,) —> invalid syntax   
delim\_t        string($”delim”)                                        string(‘,’)  —> valid syntax   
  
Notice that we reference the value of t\_date (earlier parameter value) without a $.   
That is the correct syntax.   
\*Inside\* the $[ ], the DML engine acts as if you are writing expressions inside a DML function. And that the parameter values defined earlier in the graph & sandbox are variables inside that function.   
  
  
Example of a reformat function:   
out :: reformat (in) =   
begin   
        let t\_date = “20151026”; // this is an implied data type — it automatically guesses “string”   
                                                                  // based on the initial value of “20151026”   
  
        out :: date\_month((date(“YYYYMMDD”))t\_date);   
end   
  
Notice that in the transform, I refer to the value of the variable by simply writing t\_date.   
That’s how parameter references work \*inside\* $[ ].   
  
  
Same example using parameters:   
Name                        Value                                                                Resolved Value   
t\_date                        “20151026”                                        “20151026”   
out                                $[ date\_month((date(“YYYYMMDD”))t\_date) ]        10   
  
Notice that the expression that I used to compute the output for the function can be used exactly (with no changes to the syntax) to compute the value of the parameter inside $[ ].   
  
  
  
Writing record formats & transforms is going to require that you work with quotation marks.   
— in a record format or transform, string constants are always enclosed in quotation marks.   
  
string(“,”) name = NULL(“”);   
out.field1 :: “x”;   
  
“,”, “”, and “x” are all string constants   
        default values for fields in a record format   
        delimiter values for fields in a record format   
        giving a string constant as the value for a field in a transform   
all of these are situations where you use string constants   
  
With metaprogramming, we are going to be working with string constants — as we create/modify record formats and transforms.   
  
One way to add quotation marks around a parameter value reference:   
        $”delim”   
if the resolved value of delim is |   
        then $”delim”   
resolves to “|”   
  
If you write   
        “$delim”   
then the resolved value is “$delim”   
The quotation marks on the outside of the $ prevent the $-reference from being resolved. This is a way of specifying constant interpretation when you have a $ in your string.   
  
Another way to add quotation marks around a value is to concatenate them:   
        $[ “‘“ + delim + “‘“ ]   
I am using two different sets of quotation marks. In both situations, I am writing double quote-single quote-double quote (“ ‘ “). The double quotes are because I am writing a string constant. The single quote is the string value.   
If the value of delim is |, the resolved value would be:   
        ‘|’   
  
You can do it the other way as well:   
        $[ ‘“‘ + delim + ‘“‘]   
Here I am using single quote-double quote-single quote (‘ “ ‘). The single quotes are because I am writing a string constant. The double quote is the value of that string.   
If the value of delim is |, then the resolved value would be:   
        “|”   
  
  
Another way to work with quotation marks is to escape them. (Above I used two different types of quotation marks to distinguish between the string value versus the quotation marks around the string value.) Here I’m going to use the same type of quotation mark, but I’ll escape the string value with \ (backslash). The backslash is called an “escape” because it tells the DML engine to ignore the special character (quotation mark) and treat it as a simple string.   
        $[ “\”” + delim + “\”” ]   
  
I wrote double quote-backslash-double quote-double quote (“ \ “ “). The outer quotes are indicating that we have a string constant. The value of that string constant is \”  — the backslash is there to escape the quote and treat it as a string constant rather than a special character.   
Resolved value, if delim = |, is   
        “|”   
  
This also works with single quotes (‘ \’ ‘)   
        $[ ‘\’’ + delim + ‘\’’ ]   
resolves to   
        ‘|’   
  
In DML syntax for record formats & transforms, you can use either single quotes or double quotes for string constants. So any of the above would be correct.   
  
Extremely common design pattern for record formats is to set up the begin-end of the record format as string constants and then compute the fields inside that hardcoded string   
record   
        $[ string\_join(for (let i, i < 5) : “string(‘,’) field” + (string(“”))(string(“”))i + “;”,   
                           “\n  “)]   
end   
  
===================================   
  
There are two types of script generation — you might see the earlier type if you are working with older graphs.   
        Version 2.13 compatible   
        Dynamic   
  
In order to use PDL interpretation, your script generation for the graph must be set to “Dynamic”   
If you are editing a graph and you don’t see “PDL” available as a possible Interpretation for a parameter, check the Script Generation setting:   
        Settings > Graph Settings > Script   
If you need PDL, change the generation to “Dynamic” — this may (not always) cause changes in how the parameter values are resolved and in the behavior of the parameters, so you should retest the graph if you make that change. (There are some rare side effects.)   
  
Starting in V2.14, dynamic script generation was available but not the default   
Starting in V3.0, the default script generation for new graphs is now “Dynamic”   
Starting in V3.1, the default interpretation for parameters in new graphs is PDL   
  
====================================   
  
In DML, there is syntax for a conditional expression — you see conditional expressions typically used   
        — select\_expr parameter in FILTER BY EXPRESSION   
and           
        — as the condition for if-logic                if (<conditional expression>) …   
  
  
You build a conditional expression using comparison operators (such as ==, >, <, >=) or unary operators (such as !, not) or using a function that returns a boolean (true or false)   
In Ab Initio, we don’t have a boolean data type. Instead, we use integer values   
        1        —> true   
        any other number (including 0) —> false   
  
If you have syntax that requires a conditional expression, it will return either 1 (true) or 0 (false)   
  
Notice that in our exercise (ex1), the Korn shell command returns 1 or 0   
        $( if [ $ARCHIVE = “yes” ]; then print 1; else print 0; fi ]   
  
So we can replace this with a simple conditional expression in DML and it will automatically return 1 or 0. The equality comparison operator in DML is ==   
        $[ ARCHIVE == “yes” ]   
and this will have a value of 1 (true) or 0 (false)   
  
  
In DML — there is no boolean data type.   
In parameters — there is a Boolean type Attribute. If you set the parameter Type to Boolean, then the parameter can have 1 of 2 values: True or False.   
  
Parameters do have Types — has two effects on the parameter:   
        — validates the resolved value that is computed as valid for that Type and there’s an error if   
                it’s not valid   
        — which Editor is used to edit the value if you click the Edit pencil in the Parameters Editor.   
  
For example, if you set the Type of a parameter to be “Record Format” then the resolved value of that parameter must be a valid DML record format (valid syntax) AND if you click the Edit pencil, it will open a Record Format Editor!   
  
  
Other attributes for parameters:   
        Input — either checked (default) or unchecked   
                                checked — we call this an “input parameter” and the end user is expected to provide a   
                                value for the parameter at runtime either via command-line, pset, or the GDE Input   
                                Values Editor   
                                unchecked — we call this a “local parameter” and the graph developer is expected to   
                                provide a value that will be resolved at runtime. Value for the parameter is saved as   
                                part of the graph (in the .mp file)   
  
        Required — either checked (default) or unchecked   
                                checked — end user or graph developer (depending on whether it is an input param   
                                or not) is required to provide a \*NON-BLANK\* value for the parameter   
                                unchecked — okay (not an error) for the parameter’s value to be blank.   
                                in either case, the resolved value can be blank and there is no error   
                                in the Korn shell, if a parameter’s value is blank — that parameter is “unset” — it is   
                                        regarded as the parameter not having a value at all.   
                                this is a way to make providing a value for a parameter optional (unchecked)   
  
        Export to Environment — either checked or unchecked (default)   
                                checked — parameter + value are exported into the Korn shell environment   
                                unchecked — parameter + value are local to the graph subshell only.   
                                Most of the time you want the default (unchecked = not exported)   
                                (see background below)   
  
        Kind — Keyword (default), Environment, Implicit, Positional   
                        specifies how you want to provide a value for Input parameters when you run the graph   
                        from the command line   
  
                        Keyword —                 air sandbox run my\_graph.mp -PARAM\_NAME value   
                        Positional         —         air sandbox run my\_graph.mp value   
                        Environment —         export PARAM\_NAME=value   
                                                                air sandbox run my\_graph.mp   
  
                        Keyword is a superset of Environment. If you do not specify a value for the parameter   
                        on the command-line, Co>operating System checks the environment to see if it is set   
                        there. Almost 100% of the time — you want to use default, which is Keyword.   
  
                        Positional is not recommended. There only for backward compatibility. It’s confusing &   
                        difficult to use if you have optional (= not required) or environment parameters.   
                                  
                        Implicit is used internally for our component parameters. Should not be used for graph   
                        parameters.   
  
        Location — we’ll talk about later this week & we’ll do some demo + exercises with it.   
  
—> Background before we can explain what Export to Environment does.   
When you run a graph, Co>Operating System starts a subshell. Graph runs inside that subshell. Any parameters are visible only to the graph running in that subshell. Most of the time that’s all you need. But sometimes the parameter values need to be visible to other programs (outside of the graph’s subshell). Typical cases for that are:   
        — when you are passing the parameter value as an argument to an external command   
                (like a command that you run using Run Program component)   
        — when you are overriding the value of a Co>Operating System configuration variable   
                        such as AB\_REPORT, AB\_JOB, AB\_SUMMARY\_FILE, etc. Anything that starts with AB\_   
        — when you have a parameter reference inside an external file that is saved on disk (for earlier   
                versions of the Co>Operating System before PDL interpretation and dynamic script   
                generation was available)   
In these cases, the parameter & its value must be exported to the environment so that it can be “seen” by the Korn shell or the external program.   
  
—> You have probably heard that you should \*NOT\* check “Export to Environment” unless you are sure that you need it. Why?   
For all parameters that are exported to the environment, we have to add the parameter name + value (its “definition”) to a command line command that we use to run the graph. Some OS have a limit on how many characters you can use in a command-line command. If you export a lot of parameters, you might hit that limit & you can’t run the graph! The OS fails the command.   
Also, exporting a lot of parameters to the environment slows down the graph startup somewhat.   
  
  
  
Requirement for a parameter is that is has a single non-NULL value.   
        — parameter values cannot be NULL   
        — value can be “scalar” or “compound”   
                        scalar value is a simple value like 1, -1, 0, “”, “xyz”, “20151026”   
                                        strings, decimals, integer, date/datetime   
                        compound value can be either a vector or a record   
  
If you want to hold multiple values in a single parameter, you would use a compound type like vector or record. (These are \*not\* parameter types — instead, the parameter Type is set to String. Co>Operating System recognizes record & vector syntax and allows that as String for parameters. Same thing with dates & decimals. There is an Integer parameter type.)   
  
        vector — list of values that all have the same DML type   
        record — collection of fields, each field can be a different type   
  
        you can have a vector of records (very very common type used in metaprogramming)   
  
========================================   
  
Syntax for declaring a vector — in metaprogramming we almost exclusively use length-prefixed vectors.   
        let string(10)[integer(4)] field\_names = allocate();   
  
“let” is declaring a variable inside a transform or begin-end expression   
string(10) is the data type for each element in the vector   
[integer(4)] is the length-prefix. What is the maximum value for integer(4)?   
        integer(1)                        127   
        integer(2)                        32,000   
        integer(4)                        about 2 billion   
if you use integer(4) as your prefix, your vector can have anywhere between 0 and 2 billion elements. Vector is dynamically sized based on its contents. So if it’s empty, the size is 0. If you put in 20 elements, the size is 20. Maximum size would be 2 billion.   
allocate() — when you declare a vector, you need to allocate some initial memory for its storage.   
  
        let record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end [int] veggies = allocate();   
  
“let” is declaring a variable named “veggies”, and we’re allocating with allocate() some initial memory.   
What is the data type for the elements in this vector? record, specifically:   
        record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end   
The size of the vector is specified with “[int]” —> length-prefixed vector. “int” is a shortcut for “integer(4)”   
Shortcuts are:   
        integer(1)                        —> char   
        integer(2)                        —> short   
        integer(4)                        —> int   
        integer(8)                        —> long   
  
You could write this:   
        decimal(“”)[char] num\_list;   
This is a length-prefixed vector. Each element in the vector is a delimited decimal — delimited with the NUL char (also written as “\0”). And the size of the vector can vary dynamically between 0 and 127 elements.   
  
If you’ve ever done any programming in C, you’ll have heard of NUL-terminated strings — that’s a common data type to use in C. In Ab Initio DML, you can write a NUL-terminated string (or decimal or date) as:   
        string(“”)   
or   
        string(“\0”)   
Common standard for described a delimited (variable-sized) data type.   
  
  
We have declared this type:   
        let record   
                string(int) variety;   
                integer(4) num\_plants;   
                date(“MMM-DD”) sowing\_date;   
        end [int] veggies = allocate();   
  
        veggies = [vector [record variety “Sungold” num\_plants 2 sowing\_date “JAN-03”],   
                                                [record variety “Brandywine” num\_plants 1 sowing\_date “JAN-05”],   
                                                [record variety “Jubilee” num\_plants 5 sowing\_date “JAN-01”],   
                                                [record variety “Mr. Stripey” num\_plants 3 sowing\_date “JAN-02”]];   
  
I put some values in this variable. And now I want to access the value of the 3rd element in the vector. What do I write?   
        veggies[2]   
the result is —> [record variety “Jubilee” num\_plants 5 sowing\_date “JAN-01”]   
Remember that vectors are indexed starting at 0.   
  
How do I access the value of the “variety” field in element at index 2 from veggies?   
Use dot syntax to access the value of a field inside a record (in.field1, out.cust\_id, lookup(“Some Lookup”, in.id).manager))   
        veggies[2].variety   
the result is —> “Jubilee”   
  
============================================   
  
This is a type:   
        utf8 string(int)   
charset = utf8   
form = string   
size = int   
  
A field consists of:   
        a type   
        a field name   
        optionally allowing NULL values   
        optionally having a default value   
        optionally having a condition   
        optionally having a comment   
  
This is a field:   
        utf8 string(int) address = NULL(“”) // here is a comment;   
        string(“,”) name;   
                this has an implied charset — on Linux/Unix/Windows it is ascii   
  
A record format consists of zero or more fields. Starts with the word “record” and it ends with the word “end”.   
        record   
                // fields listed here   
        end   
  
  
======================================   
  
In my graph….   
you can try this yourself in the Parameters Editor — create  new graph in BDS\_construction sandbox.   
  
store\_rec\_format                $[ record\_info (read\_type (AI\_DML + “/store.dml”)) ]   
type\_of\_first\_element        $[ store\_rec\_format[0].dml\_type ]   
last\_element                                $[ length\_of(store\_rec\_format) - 1]   
type\_of\_last\_element        $[ store\_rec\_format[last\_element].dml\_type ]   
  
In the computation for store\_rec\_format, I used the read\_type() function. There is also read\_file(). What’s the difference?   
        Both of them reads the contents of a file from the filesystem.   
        Result is a string containing the contents   
  
        Difference is   
                        read\_type() also validates the contents of the file to make sure that the contents are   
                                a valid record format   
                        read\_file() just reads the contents   
                        read\_transform() validates the contents of the file to make sure that the contents are   
                                a valid transform   
  
read\_type() is nice to use with record\_info() because read\_type verifies that you are really reading a record format \*before\* you try to break it up in the dml\_field\_info type.

**Day 2 :**

Record formats (and transforms & keys) are often saved as string values. Functions like read\_type() and read\_transform() return the record format/transform as a string value. Parameters that store record formats & transforms, act like those values are string values.   
  
let string(int) in\_dml = “record string(10) fname; string(10) lname; end”;   
  
  
Three functions to add & remove fields from a record format.   
        remove\_fields (string rec\_format, string fieldname)   
OR   
        remove\_fields (string rec\_format, [vector <list of fieldnames>])   
  
        add\_field (string rec\_format, string fieldname, string dml\_type, [optional args for other attributes for field])   
  
        add\_fields (string rec\_format, dml\_field\_info\_vec fields)   
                dml\_field\_info\_vec is a built-in type — a vector of dml\_field\_info records   
                you can create a dml\_field\_info record by using the make\_field() function   
  
  
There are two ways to represent a collection of fields (and/or record format)   
        — as a string   
        — using a structured dml\_type, such as dml\_field\_info   
Some metaprogramming functions use strings as input arguments & return strings as output. Other metaprogramming functions work with the structured types.   
  
Search Help Library for “Metaprogramming tasks using DML functions”   
        This categorizes the metaprogramming functions according to what you want to   
        do: work with data types, work with keys, work with transforms, etc.   
        Help definitions for each function will usually suggest other functions that you   
        might need to use   
  
  
  
=========================   
  
I have a delimited type:   
parameter         value           
in\_type                 “string(‘,’)”   
delimiter                $[ type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})].value ]   
  
$[ type\_info(in\_type).attributes ] —> returns a vector of records that contains various attributes for the type. There is no guarantee that there is a delimiter record and there is no guarantee on the order (you can’t assume that the delimiter attribute is always first — for example).   
  
We need to search this vector to see if there is a delimiter attribute & if yes, return the value of that delimiter attribute. Use a vector function, like vector\_search(), to find the element in the vector that has the delimiter key and return the value from that record.   
The 2nd argument for vector\_search() needs to be the same data type as the elements in the vector.   
In the “attributes” vector that type\_info() returns, the elements in the vector are records with two fields: key, value.   
The 3rd argument for vector\_search() is optionally a key that specifies which field(s) you want to match on.   
  
This expression:   
vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})   
returns the element # of the record that matches the 2nd argument based on the “key” field value.   
  
We have to access that element of the original vector   
type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})]   
And we want the value field from that element   
type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})].value   
  
Extremely helpful to build up these expressions step by step & test as you go.   
Extremely helpful to make each step be a separate parameter so that you can easily debug one step at a time. Then when you’ve the thing working —> combine all the steps into a single parameter value.   
  
Problem — in the expression above, if I provide an initial type (in\_type) that doesn’t have a delimiter, the expression fails with an error. Error is occurring because vector\_search returns -1 (no match found) and -1 cannot be used as the index into the vector.   
We can use functions such as   
        null\_if\_error()   
takes multiple arguments and it starts with the first expression and evaluates it. If it returns an error, it tries the next expression. If none of the expressions return a non-error value, then the function returns NULL.   
        null\_if\_error (complicated\_vec\_thing, “unknown”)   
  
Here’s what the final expression would look like:   
null\_if\_error(type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})], [record key “delimiter” value “no delimiter”]).value   
  
Very complicated & it repeats some parts — type\_info(in\_type) is repeated twice.   
This might be a good case for using block expression. Enables you to declare variables inside an expression and then return a single result.   
The $[ ] syntax requires that the code inside the $[ ] is a valid DML expression   
        expression —> code that returns a single result (scalar like string/date/decimal or compound type like record or vector)   
  
Basic syntax for a block expression   
$[ begin   
   <you can do whatever you want — declare local variables, write loop expression & statements, if logic…>   
   result :: <value that the block expression returns>;   
end ]   
  
  
We can take this very complicated expression   
$[ null\_if\_error(type\_info(in\_type).attributes[vector\_search(type\_info(in\_type).attributes, [record key “delimited” value “”], {key})], [record key “delimiter” value “no delimiter”]).value ]   
  
and turn it into a much simpler block expression:   
$[ begin   
        let dml\_type\_info in\_type\_info = type\_info(in\_type);   
        let record   
                string(int) key;   
                string(int) value;   
        end dummy\_record = [record key ‘delimited’ value ‘NULL’];   
        let int index = vector\_search(in\_type\_info.attributes, dummy\_record, {key});   
  
        result :: if (index >= 0) in\_type\_info.attributes[index].value   
                                else dummy\_record.value;   
end]   
  
  
Block expression is simpler than the single expression   
AND you can use the PDL debugger to step through it to debug it.   
In the Parameters Editor   
        Debug > Debug PDL   
  
  
The member operator only works for vectors.   
If you want to find if a particular string is embedded in some other string, use string\_index   
        string\_index (str1, “test”)   
The result of string\_index can be used as a conditional expression because it returns 0 (false) if the given string is not found in the original string. Returns 1 or some other number if the given string is found — 1 or some other number is equivalent to true.   
  
However, starting in version 3.2.5, we have slice expressions that work on both strings & vectors.   
        you can write something like this:   
                vec\_value [2:4]        —> returns a new vector containing elements 2, 3, 4 from the original vector   
                string\_value[2:4] —> returns a new string which is the equivalent of string\_substring(string\_value, 2, 3) — chars 2, 3, and 4 from the original string.   
    
  
=================================   
  
Let’s say that we have a setup like this:   
        your graph has three input parameters   
                in\_dml                —> input dml record format, as a string value   
                in\_delim        —> a delimiter that is used in the in\_dml   
                out\_delim        —> delimiter you want to use to replace in\_delim in in\_dml   
  
        you write this to do the replacement   
                out\_dml                $[ string\_replace (in\_dml, in\_delim, out\_delim) ]   
  
We want to replace every occurrence of the in\_delim with the specified out\_delim.   
  
This works perfectly fine if the in\_delim is something like “|”.   
Can you think of a situation where it wouldn’t work???? What’s a delimiter that would cause problems??   
        if in\_delim was a value such as “,” “;” “-“ “\_” any alphabetical char would be an issue, etc.   
        these characters are all used in DML syntax to define fields   
  
        if you had this:   
                string(“;”) fieldA;   
  
        and you specified in\_delim as “;” —> when you replaced with “,”, the result would be:   
                string(“,”) fieldA,   
        invalid DML syntax!!   
  
Using string\_replace is a very simple way to do the task (replace one delimiter with another) but it works in only limited circumstances.   
Should you use it?   
        — yes if you know what sort of delimiters the input record formats will be using   
        — no if you don’t know   
Use the simplest technique possible as long as you have some idea that it will work for the inputs that you expect to get.   
  
==============================================   
  
I want to add a new field to the \*beginning\* of an existing record format.   
The problem is — add\_field() and add\_fields() adds the new field to the \*end\* of the existing record format.   
  
How can we add a field to the beginning of an existing record format? join\_types()   
        join\_types takes 2 arguments, both of which need to be records.   
  
The existing graph has 4 input parameters:   
        input\_file   
        input\_dml   
        new\_field\_name   
        new\_field\_type   
  
I want to create a new record format that has 1 field — specified by the new\_field\_name and the new\_field\_type parameter values.   
Two methods (string manipulation OR metaprogramming functions) to do it:   
(1) string manipulation — I can just concatenate the parts together to form a simple record format   
  
Notice that I wrote out the record format syntax   
record   
        new\_field\_type new\_field\_name;   
end   
and then I put quotation marks around the things that are string constants (things that are \*not\* variables or parameters) and I use concatenation (+ operator or string\_concat() function) to put everything together   
“record “ + new\_field\_type + “ “ + new\_field\_name + ”; end”   
  
OR   
(2) metaprogramming functions   
        add\_field (“record end”, new\_field\_name, new\_field\_type)   
  
OR (if you had multiple fields this would be better)   
(3) structured metaprogramming functions   
        add\_fields(‘record end’, [vector make\_field(new\_field\_name, new\_field\_type), ….])   
  
================================   
  
Exercise 2   
Add field at the end — we can just use add\_field() — because it puts the new field at the end of the record format.   
$[ add\_field (input\_dml, new\_field\_name, new\_field\_type) ]   
  
Make the fieldnames lowercase   
$[ string\_downcase(input\_dml) ]   
        this works because all DML syntax is lowercase (data types, “record”, “end”, etc) and because none   
                of the fields use default values that have uppercase letters in them   
        the opposite (making the fieldnames uppercase — would not work, you couldn’t just use string\_upcase   
                because DML syntax like “RECORD” and “END” is invalid.   
        if any of your fields had default values that were uppercase strings — using string\_downcase() would   
                not be good.   
  
Remove the newline field   
$[ remove\_fields (input\_dml, “NEWLINE”) ]   
        “NEWLINE” is a string constant — it is not a variable or parameter reference   
        so it must have quotes around it, just like any other string constant.   
  
When we try the other pset (Run > Select Input Values —> chose pset/02b.add\_fields.pset)   
        this fails because there is no field named NEWLINE in the record format from the new pset.   
        in the new pset, the record format has a field named “newline”   
  
How do we fix this? We want to remove any fields named “NEWLINE” or “newline” — whatever is in the record format.  One way is to downcase the input\_dml before applying remove\_fields… This is the simplest (and thus an excellent idea) way that works for this particular requirement:   
$[ remove\_fields (string\_downcase(input\_dml), “newline”) ]   
This will still fail if there is no field that has a name like newline.   
  
Another way — if you were trying a bunch of different options — more general way to solve this issue is to use null\_if\_error()   
$[ null\_if\_error (remove\_fields(input\_dml, “NEWLINE”), remove\_fields(input\_dml, “newline”), remove\_fields(input\_dml, “Newline”), input\_dml) ]   
This works even if there is no newline field at all in the record format!   
  
Even better — combine both techniques…   
$[ null\_if\_error (remove\_fields (string\_downcase(input\_dml), “newline”), input\_dml) ]   
  
  
+++ For down casing the fieldnames +++++   
The more general method that always works is to loop over the vector, downcase the fieldnames only and create a new record format from that. This is fairly complicated.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let i, i < length\_of(input\_rec\_info))   
                input\_rec\_info[i].name = string\_downcase(input\_rec\_info[i].name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
  
  
Important thing   
        — you need to know what you want as output   
        for the block expression above, I wanted a record format (string) as output   
        — you need to know what the functions you are using will return   
        record\_info returns a vector of records — that is \*NOT\* a record format   
        — at the end, I know that I can add that vector of records into a string record format using the   
                add\_fields() function   
  
  
I wrote this index for loop statement which works correctly to downcase the names of the fields in a record format. This method here (immediately below) is the best performance that gets the correct result.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let i, i < length\_of(input\_rec\_info))   
                input\_rec\_info[i].name = string\_downcase(input\_rec\_info[i].name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
  
I started to write this — and then realized that it wouldn’t work. This is an element for loop statement.   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        for (let field in input\_rec\_info)   
                field.name = string\_downcase(field.name);   
                  
        result :: add\_fields (“record end”, input\_rec\_info);   
end ]   
This doesn’t work — the output record format is the same as the input record format — the field names are not downcased, they are still uppercase.   
This cannot work because of this part “let field” —> creates a variable instance that contains a record for the current field. In the loop, we are changing the name of the field \*inside\* that variable instance to be downcase, but we are \*NOT\* changing the name of the field inside the vector.   
  
  
There is one way to get this to work:   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        let dml\_field\_info\_vec output\_rec\_info = allocate();   
  
        for (let field in input\_rec\_info) begin   
                field.name = string\_downcase(field.name);   
                output\_rec\_info = vector\_append(output\_rec\_info, field); // running vector\_append inside a loop is slow   
        end   
                  
        result :: add\_fields (“record end”, output\_rec\_info);   
end ]   
  
Another way to get this to work:   
$[ begin   
        let input\_rec\_info = record\_info(input\_dml);   
        let string(int) output\_dml = “record end”;   
  
        for (let field in input\_rec\_info) begin   
                field.name = string\_downcase(field.name);   
                output\_dml = add\_field(output\_dml, field); // running add\_field inside a loop is slow   
        end   
                  
        result :: output\_dml;   
end ]   
  
The problem with both of these methods is that they are significantly slower than the method I showed you at the beginning (use an index for loop and add the fields at once, at the end).   
  
+++++   
Performance tip: minimize the amount of work that you do inside a loop. The instructions inside the loop should be as simple as possible.   
+++++   
  
  
Bonus question — how would you add a field in the \*middle\* of an existing record format?   
Let’s say that we want the end user to choose which position to add the field into. Position 1 = first field in the record format, position 3 = third field in the record format, etc.   
  
Need to declare input parameters   
        input\_dml   
        new\_field\_name   
        new\_field\_type   
        new\_field\_position   
  
Do you think this will be easier to do working with the record format as a string or as a vector?   
The problem with working with the record format as a string is that we don’t know what to search for in the record format so that we can find the position where we want to insert the new field. If we reframed the question to be — how can we add a new field after an existing field in the record format — AND the end user gives us the name of the existing field — then we might be able to do this as a string.   
  
But without knowing where in the string to go or what field name to search for in the string, using the record format as string doesn’t work.   
  
We want to work with the fields in the record format as a vector. Now the fields are indexed in the dml\_field\_info vector and we can insert the new field into that vector.   
  
You can use looping syntax to insert a new element into position in an output vector — but, what you have to do is check the index with a if-statement to know when you’re at the right position & then insert two elements at that point — the new element and then the existing element that was already at that position. Complicated.   
  
Become familiar with the \*vector\* functions that are built-in to DML. There are lot of things you can do with vectors that don’t require looping.   
There are vector functions that are similar to what you can do with string functions.   
  
Let’s reframe the question again. What if I had a string value “abcdef” and I wanted to insert “X” at position 3. How would you do that with string functions?   
        string\_substring(str, 1, position-1) + “X” + string\_substring(str, position, length\_of(str))   
  
Our record format problem is extremely similar to this — except that we have elements in a vector instead of characters in a string. Vector functions can help us:   
        vector\_slice —> equivalent of string\_substring   
        vector\_concat —> equivalent of + (concat)   
  
        in this version of the Co>Operating System vector\_concat takes only 2 arguments. (Most recent version   
        3.2.5 allows more).   
        vector\_append — appends a value to a vector   
  
        vector\_slice is not exactly the same as string\_substring   
                string\_substring (str, starting\_position, length)   
                vector\_slice (vet, starting\_position, ending\_position)   
        Also vectors start numbering at position 0 and strings start number at position 1.   
  
  
$[ begin   
        let in\_rec\_info = record\_info(input\_dml); // input\_dml as a vector of records (each record is a field)   
  
        result :: add\_fields(“record end”, vector\_concat(vector\_append (vector\_slice(in\_rec\_info, 0, new\_field\_position-2), make\_field(new\_field\_name, new\_field\_type)), vector\_slice(in\_rec\_info, new\_field\_position-1, length\_of(in\_rec\_info)-1));   
end ]   
  
Remember that you can build this more simply using multiple parameters or variables inside your block expression so that you can test each part as you go.

**Day 3 :**

Design pattern for metaprogramming:   
        end user provides a record format for the source data   
        use metaprogramming functions to compute the other record formats used   
                in the graph   
        making modifications to the source record format   
                — adding or dropping fields   
                — modifying the field names   
                — changing the data types   
  
Design pattern for metaprogramming:   
        end user provides a record format for the source data   
        use metaprogramming functions (and often looping expressions & statements) to compute transform rules for the transform components in the graph   
        looping through the fields in the source record format & adding rules to:   
                — aggregate some of the fields (sum, count, min, max…)   
                — modify the values of some fields   
                — assign fields to the output record format that you computed also with   
                        metaprogramming functions   
  
  
Here’s a very simple transform function signature:   
out :: reformat (in) =   
begin   
end   
  
But there’s a lot more that you can do… you can have multiple arguments (like for a Join transform function), the arguments can have data types.   
  
You have the choice of using string functions to create the function signature…   
let string(int) xfr = “out :: reformat (in) =   
begin   
end”   
  
Or you could use metaprogramming functions in cases where the function signature is more complicated:   
let string(int) xfr = make\_transform(“join”, [vector make\_arg(“in0”, “record”), make\_arg(“in1”, “record”)] );   
  
  
Metaprogramming tip:   
        write out an example (hardcoded) of the record format or transform that you want to create   
        decide which parts are hardcoded string constants —> enclose those in quotes   
        replace everything else with the appropriate parameter or variable references   
  
  
Let’s say that I have an input parameter named:   
        new\_field\_name   
  
I want to create a transform function that looks like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.new\_field\_name :: “X”;   
end   
  
In metaprogramming, what I would write is this:   
‘out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.’ + new\_field\_name + ‘:: “X”;   
end’   
  
I could also do this:   
let string(int) xfr = ‘out :: reformat (in) = begin out.\* :: in.\*; end’;   
xfr = add\_rule(xfr, ‘out.’ + new\_field\_name, ‘“X”’);   
          
        single quote — double-quote — X — double-quote — single-quote   
        single quotes are because “X” is a string constant inside the add\_rules function   
        double quotes are because X is a string constant inside the transform that we want to create   
                with the metaprogramming functions   
  
You could write this with the quotes in the other order   
        double quote — single quote — X — single-quote — double-quote   
like this:   
let string(int) xfr = ‘out :: reformat (in) = begin out.\* :: in.\*; end’;   
xfr = add\_rule(xfr, ‘out.’ + new\_field\_name, “‘X’”);   
The order of the quotes — whether you use single or double quotes for the outer quoting — doesn’t matter. All that matters is that you’re consistent — all the single quotes are on the outside & double quotes on the inside — or vice-versa.   
  
  
If I want to write a rule that looks like this:   
        out.new\_field\_name :: string\_trim(in.new\_field\_value);   
  
and new\_field\_name & new\_field\_value are input parameters, then my add\_rule function looks like this:   
        xfr = add\_rule(xfr, “out.” + new\_field\_name, “string\_trim(in.” + new\_field\_value + “)” );   
  
The add\_rule function automatically adds the “::” assignment operator and the “;” semi-colon that ends the rule.   
In your add\_rule function, you write only the code necessary to describe the left-hand side (lhs) and the right-hand side of the rule (rhs).   
  
==========================   
  
We are extracting data from a mainframe and loading it into a database table.   
The DML record format for the mainframe data looks like this:   
record   
   string(10) FIELD1;   
   string(10) FIELD2;   
   string(20) FIELD3;   
end   
  
Typically mainframe record formats use capital letters for the field names and they are usually fixed-size fields.   
  
The DML record format for the database table looks like this:   
record   
         string(“\x01”) field1;   
        string(“\x01”) field2;   
        string(“\x01”) field3;   
        string(1) newline = “\n”;   
end   
  
Typically record formats generated for database tables use lower case field names, and usually have delimited fields and a newline field at the end.   
  
We want to use metaprogramming to create a transform to transform the mainframe data format into the database table format.   
  
How do we approach this problem?   
1- Write out examples of what your inputs might look like — we have that above.   
2- Write out an example of what your result should look like, based on those inputs.   
3- Code up your metaprogramming to create that result (enclose string constants in quotes, concatenate with the parameterized values)   
4- Test with different inputs to check that your code is generic enough to work correctly in many situations.   
  
in\_dml =                                                                         out\_dml =   
record                                                                                record   
   string(10) FIELD1;                                              string(“\x01”) field1;   
   string(10) FIELD2;                                                string(“\x01”) field2;   
   string(20) FIELD3;                                                string(“\x01”) field3;   
end                                                                                        string(1) newline = “\n”;   
                                                                                        end   
  
I want to write a transform to transform the data from the in format to the out format.   
I also want to make sure to trim the strings from the input (string\_trim)   
I want the result to be a transform that looks like this:   
out :: reformat (in) =   
begin   
        out.field1 :: string\_trim(in.FIELD1);   
        out.field2 :: string\_trim(in.FIELD2);   
        out.field3 :: string\_trim(in.FIELD3);   
end   
Do I need a rule to assign to the newline field in the output? No — it has a default value in the record format for the output, so it will just pick up that default value.   
  
Obviously I want a loop here to create the rules for the transform. Should I loop over the in record format fields or the out record format fields?   
If I loop over the output, I have to remember to skip the newline field because we don’t need a rule for it.   
In \*this case\* looping over the input is a better idea. (Most of the time, looping over the output fields is a better idea because — to have a valid transform, every output field must be assigned a value.)   
  
$[ begin   
        let string(int) xfr = make\_transform (“reformat”, [vector make\_arg(“in”)]);   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let dml\_rule\_vec out\_rules = for (let fname in in\_fields) :   
                                make\_rule (“out.” + string\_downcase(fname), “string\_trim(in.” + fname + “)”);   
          
        result :: add\_rules(xfr, out\_rules);   
end ]   
  
(You could also write this code by using add\_rule() inside a for loop statement — instead of what I did [use make\_rule inside a for loop expression] — but that will be much slower than what I did. Avoid using add\_rule() or add\_field() inside loops. Much better to create a vector of fields with make\_rule() or make\_field() and then use add\_rules()/add\_fields() outside the loops.)   
  
What are the metaprogramming functions that we learned yesterday?   
        type\_info — extracts information from a data type like “string(5)”   
        read\_type — reads a record format from a file on disk and returns it as a string   
        record\_info — extracts information from a record format into a vector of fields   
                                                containing the field information (name, dml\_type, comments, etc.) for each   
                                                field   
        record\_info\_item — extracts one attribute of the fields from a record format into a vector   
                                                example: extract the names of all the fields in the record format   
                                                        OR extract the DML types of all the fields in the record format   
  
I want to loop over the names of the fields from the input record format.   
        record\_info\_item   
  
=============   
Exercise 3   
Hint — you want the finished transform to look like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.new\_field\_name :: ‘X’;   
end   
  
Except that “new\_field\_name” should be replaced with the actual field name selected as the input parameter value by the end user.   
  
In yesterday’s exercise, we added a new field to the output record format.   
Output record format = Input record format + new\_field\_name   
  
In today’s exercise, we want to create a transform for transforming the data from the input record format to the output record format. The transform needs to have a wildcard rule (for all the fields that are the same in both in & out) plus a rule to assign a value for the new output field.   
  
Approach #1 — string manipulation — works well in this situation because there is only 1 parameter/variable reference in the transform (to new\_field\_name). Very simple to do.   
$[ “out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
        out.” + new\_field\_name + “:: ‘X’;   
end” ]   
  
  
Approach #2 — use metaprogramming functions   
$[ begin   
        let string(int) xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
        xfr = add\_rule (xfr, “out.\*”, “in.\*”);   
        result :: add\_rule(xfr, “out.” + new\_field\_name, “‘X’”);   
end ]   
  
  
=======================   
  
        let dml\_rule\_vec  out\_rules = allocate();   
versus   
        let out\_rules = allocate(); —> this won’t work!!   
  
But this will work:   
        let out\_rules = for (let field\_name in in\_fields) : make\_rule (“out.” + field\_name, “string\_trim(in.” + field\_name + “)”);   
  
What’s the difference?   
In the second example — we are assigning a value to out\_rules using a function (make\_rule) with a known return type (dml\_rule). DML engine can figure out the data type of out\_rules based on the function that we’re using to assign a value to it. This practice is called using an “inferred type”.   
  
  
Simpler example:   
        let x = 0;   
DML engine decides that “x” is an integer because you initialized it with an integer value. Default integer type is integer(4).   
  
        let y = “abc”;   
DML engine decides that “y” is a string(3) because you initialized it with 3 character string value.   
  
  
If you wrote   
        x = x + 1;   
it would work because “1” is a valid value for an integer(4)   
  
If you wrote:   
        y = y + “x”;   
it would fail because “abcx” is \*not\* a valid string(3).   
  
Inferred types can be used \*ONLY\* if you initialize the variable with a function that returns a known type — record\_info, transform\_info — most of the metaprogramming functions do this. It’s a nice shortcut because I can never remember the built-in type names.   
  
  
In my solution for Exercise #3, I wrote this:   
        let string(int) xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
Why didn’t I use an inferred type for xfr? make\_transform has a known return type (string).   
  
If I had done this:   
        let xfr = make\_transform(“reformat”, [vector make\_arg(“in”)]);   
Because the DML engine would have created a fixed-size string with the fixed-size of the return value of make\_transform.   
That’s okay — until I try to do this:   
        xfr = add\_rule(xfr, “out.\*”, “in.\*”)   
This line would fail saying that the type for “xfr” is not large enough to add additional characters to it!   
  
Recommendation:   
Don’t use inferred types for strings — use length-prefixed strings   
        let string(int) xfr ….   
        let string(int) dml ….   
  
Do use inferred types for functions that return records such as…   
         record\_info, transform\_info, type\_info   
Do use inferred types for functions that return structured field or rule information…   
        make\_rule, make\_field, make\_arg   
  
Functions that return strings are:   
        add\_rule, add\_rules, add\_field, add\_fields   
Use string(int) as the type for variables that hold those values.   
  
========================   
  
Reading record formats & transforms that are “landed to disk” (files saved in your sandbox), two access methods:   
  
1- Use a function to read the contents of the file as a string   
        read\_type() —> reads & validates types (record formats)   
        read\_transform() —> reads & validates transforms   
        read\_file() —> general files   
  
OR   
2- Setup the parameter accordingly   
        Value        —> path to the file that you want to read   
        Resolved Value —> contents of the file   
  
        (a) Choose the Type of the parameter to be something like Record Format or Transform   
        (b) Set the Location attribute of the parameter to “File”   
  
  
Both of these methods — assuming you use read\_type() or read\_transform() — validate the contents of the file to make sure that they are a valid record format or transform.   
With the functional method (#1), you can store the path to the file as a separate parameter if you want…   
name                        value                                                        resolved value   
in\_dml\_path        $AI\_DML/f.dml                                /path/to/your/sandbox/dml/f.dml   
in\_dml                        $[ read\_type(in\_dml\_path)] <contents of the dml file>   
  
Now you can use both these parameters as inputs to other computations.   
    
If you don’t need to access both the contents of the file & the path to the file, then using the location (Method #2) is more convenient. Then you have just a single parameter — that is easy to use as an Input parameter (end user can easily browse for the file name in the sandbox) and the resolved value is the contents of that file.   
  
One thing to remember — make sure that the Location attribute is set correctly   
        if the value of the parameter is a path —> Location = File   
        if the value of the parameter is embedded record format or transform —> Location = Embedded   
  
        in both these cases, the \*resolved\* value of the parameter will be the actual record format or transform contents.   
  
Another reason to use method #2 (location) is that the user can specify how they want to provide the value at runtime…   
        they can provide a transform or record format from a file (Location = File)   
OR   
        they can type in (copy & paste…) a transform or record format directly (Location = Embedded)   
  
AND you can switch between them… so   
        if the end user starts with Location = File and choose a file   
        and then switch to Location = Embedded, the GDE will prompt the end user (just like it does when doing this on the Ports tab of a component) if you want to use the contents of that file as the embedded value. Enables end user to import the contents of an existing record format or transform and then make edits to it and use that as the value of the parameter!   
    
======================================   
  
You “declare” a parameter once. Declaration of a parameter consists of choosing a Name, a Type, whether or not it is an Input parameter, whether or not it is Required, whether or not it is Exported to Environment, what is its Kind.   
—> think of a parameter like a variable, you can declare a variable only once   
  
You can “define” a parameter as many times as you want. The Definition of the parameter consists of giving it a Value, choosing an Interpretation, and choosing a Location.   
—> think of a parameter like a variable, you can set the value of the parameter (and reset it) as many times as you want.   
  
You declare parameters in your project parameters (.project.pset) and in your graph (in the .mp file).   
You can define parameter values in the project parameters, in sandbox parameters (.sandbox.pset), in the graph (.mp file), in pset (my\_graph.pset), in the environment, and on the command-line.   
  
Because Location & Interpretation are attributes of the parameter \*definition\*, they can be changed by the end user in the Input Values Editor.   
  
  
Plans are the same as graphs —   
        you can declare parameters in the plan (my\_plan.plan)   
        and you can define parameter value + location + interpretation in the pset (my\_plan\_2.pset)   
Plans have an Input Values Editor just like graphs do.   
  
If you’re doing this from the command-line, there are command-line commands that enable you to change the interpretation/location/etc.   
        “air sandbox parameter”   
                can be used for sandbox parameters, graph parameters, and plan parameters   
                just give the path to the pset that you want to edit.   
  
======   
  
We want to find the most recently modified .dat file in the $AI\_SERIAL directory. Use this file as input to the graph.   
  
directory\_listing(AI\_SERIAL, “\*.dat”) —> vector   
loop over the elements in that vector, use file\_information() function to get the modification date, and keep track of which one has the maximum value so far.   
  
$[ begin   
   let file\_list = directory\_listing(AI\_SERIAL, “\*.dat”);   
   let integer(8) max\_mod\_date = 0;   
   let integer(8) cur\_mod\_date = 0;   
   let max\_index = 0;   
   let index = 0;   
  
   for (let file in file\_list) begin   
        cur\_mod\_date = file\_information(AI\_SERIAL + “/“ + file).modified;   
     if (cur\_mod\_date > max\_mod\_date) begin   
                max\_mod\_date = cur\_mod\_date;   
                max\_index = index;   
     end   
        index = index + 1;   
   result :: AI\_SERIAL + “/“ + file\_list[max\_index];   
end ]   
  
==========================================   
  
Question:   
I want to read in the contents of a file and extract some information from that file.   
  
Sometimes the format of the file is simple — max\_id used by some other graph.   
123   
  
Sometimes the format of the file is kind of like a lookup file — where you have a different key (and its related value) on each line.   
Example is a list of error codes   
0,no error   
1,file not found   
2,file is not readable   
3,file is too large   
  
We want the end user to specify the key value and have the value of a parameter be the related value for that key.   
parameter        value                                                        resolved   
key\_number        2                                                                <input value specified by the end user>   
error\_msg        $[ ???? ]                                                “file is not readable”   
Equivalent of doing a lookup — only you’re doing it in the parameters!   
  
CAUTION: this is an extremely bad idea if the file is large. You should design the graph to use a regular lookup.   
But if the file is quite small, this is a simple & effective way to lookup a value from that file.   
  
We know that read\_file() will read the contents of a file on disk.   
read\_file(AI\_REFERENCE + “/error\_codes.dat”) —> contents of the file as a string value   
  
Now I have a string — how can I locate a specific line in that file.   
—> convert it to a vector   
—> use something like vector\_search to locate the matching key value.   
  
I want to have this:   
error\_code\_lkup =        [vector [record key 0 error\_msg “no error”],   
                                                                [record key 1 error\_msg “file not found”],   
                                                                [record key 2 error\_msg “file is not readable”],   
                                                                [record key 3 error\_msg “file is too large”]]   
  
vector\_search(error\_code\_lkup, [record key ERROR\_CODE error\_msg “”], {key}) —> this will tell me which element in the vector matches that key   
  
error\_code\_lkup[vector\_search(error\_code\_lkup, [record key ERROR\_CODE error\_msg “”], {key})].error\_msg  —> this will give me the error\_msg for the matching key   
  
  
I want to take this string:   
0,no error   
1,file not found   
2,file is not readable   
3,file is too large   
  
and reinterpret it as this vector:   
[vector [record key 0 error\_msg “no error”],   
                                                                [record key 1 error\_msg “file not found”],   
                                                                [record key 2 error\_msg “file is not readable”],   
                                                                [record key 3 error\_msg “file is too large”]]   
  
What type should I reinterpret as?   
Each line is a comma-delimited key, followed by a newline-delimited error\_msg   
0,no error   
  
We use this record format to describe each line in the string:   
record   
  string(“,”) key;   
  string(“\n”) error\_msg;   
end   
  
Each line will become an element in the vector — but we don’t know how many elements are in the vector to start with — so we’ll use something called a “self-sized vector”   
A self-sized vector is written like this:   
        <data type> [ ] fieldname;   
  
In transforms, using length-prefixed vectors is the recommended approach:   
        <date type> [int] variable\_name;   
But in reinterpret\_as — the issue is that you don’t have a length-prefix in your data. So a self-sized vector — it keeps adding elements to the vector until it runs of out of elements, works well with reinterpret\_as.   
  
reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ], error\_code\_str)   
—> this won’t work — it will be weird output   
  
read\_file returns a length-prefixed string — length-prefix of the string is how many chars in the string. The length-prefix data type is integer(4). We need to skip over the first 4 bytes in the value that read\_file returns because that is the length-prefix and it will mess up the reinterpret\_as.   
reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ], error\_code\_str, 4)   
  
the third argument (4) is called the offset — it tells reinterpret\_as to skip over the first 4 bytes in the string and start converting to the vector after that.   
  
  
Put this all together into a single block-expression…   
$[ begin   
        let string(int) error\_msg\_str = read\_file(PROJECT\_DIR + “/ref/error\_codes.dat”);   
        let error\_msg\_vec = reinterpret\_as( record string(“,”) key; string(“\n”) error\_msg; end [ ],   
                                                                                                error\_code\_str, 4);   
        let index = vector\_search(error\_msg\_vec, [record key KEY error\_msg “”], {key});   
  
        result :: null\_if\_error(error\_msg\_vec[index].error\_msg, “KEY NOT FOUND!”);   
end ]   
  
This is a template that you can use to do simple lookups in your Parameters Editor.   
The file path that you read will be different.   
The data type that you use for reinterpret\_as might be different   
The dummy record that you use for vector\_search will depend on the data type that you used for reinterpret\_as   
Which field you choose as output in the result will also depend on your data type.

**Day 4 :**

Exercise 3 — working with files   
Open 03.RMF.mp   
  
1- Write a transform for REFORMAT that uses the directory\_listing() function to output the list\_of\_filenames output field.   
Hint: use the Help Library examples for directory\_listing() to find an example of how to write the pattern matching argument of directory\_listing() so that you do \*not\* output dot files (e.g., “.” and “..”) in the list   
  
<You should be able to run the graph at this point to test your work so far.>   
  
2- Replace the current “out” port format for READ MULTIPLE FILES. Use metaprogramming to add a new field (the filename) to the RMF\_input\_type record format.   
  
3- Modify the transform of READ MULTIPLE FILES   
        — uncomment the line “type input\_input” and edit it to look like this:   
                type input\_type = ${RMF\_input\_type};   
  
        — edit the reformat() function to add a rule to output the filename (you do not need to use metaprogramming for this)   
  
        — make sure to set the Interpretation of the transform so that it can evaluate the ${ } reference used for the input type   
  
4. Run the graph to test your work. Don’t worry if the transaction\_amt field looks weird in the output data — that’s a problem with the data that we supplied for this exercise.   
  
====================================   
  
If you plan to use READ MULTIPLE FILES to reformat the data that it reads from the files, then you have three record formats to deal with:   
        — in port record format —> which is a list of filenames,   
                typically has a record format like this:   
                record   
                        string(int) filename;   
                end   
  
        — out port record format —> which is the result of reformatting the data inside   
                        READ MULTIPLE FILES   
                in our case, the out port record format = original contents of the files + filename   
                we want to do this:   
                $[ add\_field (RMF\_input\_type, “filename”, “string(int)”) ]   
  
        — record format that describes the contents of the files as they are read \*before\* they   
                are reformatted (before the filenames are added — the filenames are \*not\* present   
                in the original files — we’re using the reformat() transform inside RMF to add those   
                filenames to the data)   
                you specify the record format for the \*contents\* of the original files using the   
                transform in RMF. You have a line like this:   
                type input\_type = ${RMF\_input\_type};   
  
===============   
  
In earlier versions of the Co>Operating System   
        the only way to get information from the database into a parameter value was to use   
        a Korn shell interpreted parameter and use a utility like “m\_db”   
  
In recent versions of the Co>Operating System   
        we have been adding metaprogramming functions that enable you to access the database   
        using functions + $[ ] syntax (PDL interpretation)   
        advantage: significantly faster than the earlier method   
                when you use the database metaprogramming functions —> we collect all the   
                ones that you are using together into a single connection to the db   
                when you use m\_db utility, each use of m\_db is a separate database connection   
  
Example:   
        $[ db\_get\_int (AI\_DB + “/training.dbc”, “select max(id) from transactions”) ]   
  
================   
  
Some parameters need context — parameters that use a record format   
        parameters that have Type set to Transform (because the transform needs an input record format + output record format to validate it!), Key Specifier (because you need to know the input record format to choose the key), etc.   
  
Context comes from the \*graph\* design.   
  
If you have a SORT component and you Export the “key” parameter of the SORT component to use the parameter name “sort\_key”   
then in your parameter editor, if you have sort\_key Type = Key Specifier, then when you edit the sort\_key value with the Edit pencil, it will show the “in” port record format from the SORT component in the graph.   
  
1- Create a parameter in the Parameters Editor, set its type accordingly (for keys — Key Specifier).   
2- In a component in the graph, choose a component parameter — such as “key” for SORT — and click the “Export” button.   
3- In the Export dialog, change the name of the parameter to match the parameter that you created in step #1.   
  
Step #2 — the export — links the parameter to the context of that component. “Context” — record formats that component uses in the graph design.   
  
Enables the end user to be able to click the Edit pencil & see a Key Specifier Editor populated with the fields from the record format of the component that uses that parameter.   
  
Shortcut   
1- Start with the component in the graph — choose the component parameter, click Export button, and create a new parameter name.   
In the Parameters Editor, a new parameter will be created with the correct Type + context.   
  
=====================   
  
Key metaprogramming functions (these are not our most consistent moment)   
  
Design patterns you might use with add\_key()   
        sort\_key = {first\_name descending; last\_name descending}   
        more\_fields = {cust\_id}   
        $[ add\_key (sort\_key, key\_info(more\_fields)[0]) ]   
  
make\_key() is different than make\_field() and make\_rule()   
        make\_field & make\_rule take string arguments and create a structured output   
        make\_field (“fieldname”, “string(int)”) —> dml\_field\_info record   
        make\_rule(“out.\*”, “in.\*”) —> dml\_rule\_info record   
  
        make\_key   
                input argument is a dml\_key\_info\_vec   
                output result is a string (key string that can be used in a component)   
  
          
Because the key functions are confusingly (and inconsistently) named and because you often don’t use modifiers in the key values, you can get away with a simple string + vector manipulation trick.   
  
Simple key with no modifiers, might look like this:   
some\_key        { first\_name; last\_name}   
  
1- Strip off the spaces and curly braces using string\_filter\_out   
                key\_filtered          $[ string\_filter\_out (some\_key, “{ }”) ]   
          
        “{ }” is left curly brace followed by a space character followed by a right curly brace   
  
2- Use string\_split to convert the string into a vector   
                key\_vec         $[ string\_split(key\_filtered, “;”) ]   
  
        result —> [vector “first\_name”, “last\_name”]   
        Now I have a vector of strings & I can manipulate it as a vector. I can add additional   
        fields…   
                key\_vec\_all          $[ vector\_append (key\_vec, “cust\_id”) ]   
        result —> [vector “first\_name”, “last\_name”, “cust\_id”]   
          
Now I might want to convert that vector back into being a key string. Reverse the steps.   
3- Use string\_join to convert the vector into a string   
                key\_vec\_all\_str   $[ string\_join( key\_vec\_all, “;”) ]   
  
        result —>  first\_name;last\_name;cust\_id   
  
4- Add the curly braces back into   
                final\_key\_str                “{“ + key\_vec\_all\_str + “}”   
          
        result —> {first\_name;last\_name;cust\_id}   
  
  
IMPORTANT: This method does not work if you have modifiers in the key.   
If you were more careful about how you remove spaces — loop over the vector elements to trim them — then it could work even if your keys have modifiers.   
  
================================   
  
The graph for exercise 4 has a SORT component in the graph that is “conditioned out” — on the Condition tab — the value of the condition is always False.   
This component is always disabled.   
  
The key parameter is set to use the parameter value $fields\_to\_drop   
$fields\_to\_drop is an Input graph parameter.   
  
This is a roundabout way to enable the end user to select one or more fields from the list of input fields.   
We’re not using $fields\_to\_drop as a key — we’re taking advantage of the fact that the Key Specifier Editor enables you to select one or more fields from the record format. We’re linking the $fields\_to\_drop parameter with a record format + editor (Key Specifier Editor) as a user-interface decision — easy way for the end user to see a list of the input fields & choose one or more of them to drop from the data.   
  
There’s a REFORMAT component in the graph — why not associate the parameter with REFORMAT?   
REFORMAT doesn’t have a key parameter. There’s no way to get the same user interface interaction (an editor that enables the end user to select one or more fields from the input record format).   
  
SORT is not the only option for the workaround. You could use any component that has a key parameter and does not transform the data —> another option would be DEDUP SORTED. Again, in both situations, the component does NOT run. It is simply there to provide a link between the input record format + graph parameter so that the end user has a nice user interface.   
  
If you didn’t care about the user interface — you don’t need this.   
You would simply set fields\_to\_drop to be a string parameter and expect that the user would enter a delimited list of fieldnames. (That is not too difficult to do — you would need to do some input validation to make sure that the user formatted the list correctly. You should document the parameter to indicate what format you want for the list of fields.)   
  
====================   
  
Exercise 4   
        two input parameters:   
                in\_dml — value is a path to a dml file, supplied by the end user   
                                        resolved value is a record format (Location = File)   
  
                fields\_to\_drop — value is a key supplied by the end user using the Key Specifier Editor   
  
  
Option #1 — string + vector manipulation to remove fields\_to\_drop from the in\_dml   
        $[ remove\_fields (in\_dml, string\_split(string\_filter\_out(fields\_to\_drop, “{ }”), “;”)) ]   
  
        even better: use null\_if\_error() around remove\_fields to do error handling   
                        remove\_fields fails if the specified fields are not found in the in\_dml   
  
Option #2 — use metaprogramming functions + loop   
        $[ remove\_fields (in\_dml, for (let field in key\_info (fields\_to\_drop)) : field.field\_name ]   
          
        disadvantage of this method: more difficult to write   
        advantage of this method: works correctly even if the key fields have modifiers on them   
        even better: add null\_if\_error() to do error handling   
  
=====================================   
  
In a typical transform (no metaprogramming being used), if you have a operation (that has several steps) that you need to do repeatedly — you might create a user-defined function and save that function in an external file (utilties.xfr, string\_cleansing\_functions.xfr, etc.)   
  
in utilities.xfr, I might create a simple function that concatenates a person’s given name & family name — but also deals with blank & NULL values.   
  
out :: full\_name (string(int) given\_name, string(int) family\_name) =   
begin   
        let string(int) fname = if (not is\_null(given\_name) and not is blank(given\_name))   
                                                                 given\_name + “ “   
                                                        else “”;   
        let string(int) lname = first\_defined(family\_name, “”);   
  
        out :: fname + lname;   
end   
  
  
How do I use this function in the transform of a component in a graph?   
Include the file where I’ve saved the function in….   
  
include “~$AI\_XFR/utilities.xfr”;   
  
out :: reformat (in) =   
begin   
        out.customer\_name :: full\_name (in.first\_name, in.last\_name);   
end   
  
  
In metaprogramming — what if we want to use a user-defined function to compute the value of a parameter?   
In a block expression   
$[ begin   
        result :: ….   
end ]   
you cannot have global variables or include statements.   
  
Solution: define the value of a parameter AB\_DML\_DEFS and include the file containing the user-defined functions or define the user-defined function as the parameter value.   
  
Value of AB\_DML\_DEFS is \*not\* a file path. It is the functions themselves — as an embedded value or the contents that are read from a file.   
  
============   
  
Really neat example of a highly generic graph that uses a significant amount of metaprogramming…   
        Help > Examples > Differencing graph   
  
Great resource to inspect for ideas on how to do things, best practices, etc.   
  
Graph compares two files of any type — and generates a report with information about the differences between the files. Very customizable — you can ignore differences in certain fields, specify how to deal with NULL values, specify how to deal with differences in element values in a vector, etc.

**Day 5 :**

In a typical design pattern — the end user supplies a source record format and chooses the key fields for the Rollup plus also choose which fields to aggregate.   
        in\_dml   
        rollup\_key   
        fields\_to\_aggregate   
  
In the “anti-rollup” design pattern — the end user supplies a source record format and choose which fields to aggregate (usually with min/max functions).   
        in\_dml   
        fields\_to\_aggregate   
The key for the Rollup is \*all\* of the remaining fields in the in\_dml that are not being aggregated. This is design pattern that analysts sometimes use — they are more interested in finding min/max values over a group than what the group key is.   
  
In Exercise 5 — we’ll be doing an anti-rollup.   
        We’ll compute the key for the Rollup component by subtracting the list of fields to aggregate from the list of fields in the input\_dml — with metaprogramming.   
  
=====   
  
Extremely common design pattern for metaprogramming — one that is used well in the exercise 5 graph (05.aggregate\_fields.mp)   
is the idea that vectors are easier to work with.   
Convert things to vectors — lists of items — and then you can use vector functions & loops to work with those lists.   
And then convert back to strings — transforms, keys, record formats — at the end.   
  
When combining multiple parameters into a single parameter computation — either using functional composition or using block expression — you only want to combine parameters that are \*not\* used elsewhere in the parameter computations.   
  
If fields\_to\_aggregate\_\_vec is used to compute both the record format and the transform —> then it needs to be a separate parameter so that you don’t have to recompute it both times.   
If aggregate\_info is used only to compute the record format, then it can be combined with the computation for the record format — it shouldn’t be a separate parameter.   
  
How can you find out where a particular parameter is referenced?   
Find!   
        Edit > Find in the Parameters Editor   
  
  
It is helpful to combine multiple computations into a single parameter using a block expression because you can use the PDL debugger (in the Parameters Editor, Debug > Debug PDL) to step through the computation, evaluate expressions in the context of the variables in the expression, and so on.   
  
========================   
  
Example use case   
  
We have an input record format (in\_dml) that is supplied by the end user as an Input parameter value, and it has a value like this:   
record   
  string(“”) first\_name;   
  string(“”) last\_name;   
  string(“”) address;   
end   
  
We want to check if the value of the input field in each record is valid & add a flag (true or false) to indicate whether or not the value is valid.   
  
We want to use metaprogramming to compute an output record format that looks like this:   
record   
  string(“”) first\_name;   
  string(“”) last\_name;   
  string(“”) address;   
  integer(1) first\_name\_is\_valid;   
  integer(1) last\_name\_is\_valid;   
  integer(1) address\_is\_valid;   
end   
  
How would you approach this problem? Any ideas??? How would you use metaprogramming to create the output record format? You’ll need to use add\_fields to in\_dml — how do you get the vector to indicate what fields to add?   
1- Extract the field names from the in\_dml in vector form. What function would we use? record\_info\_item   
2- Loop over those field names and add \_is\_valid to the name of each —> new vector.   
3- Use add\_fields to add the new vector to the existing in\_dml.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let validation\_fields = for (let field in in\_fields) : make\_field(field + “\_is\_valid”, “integer(1)”);   
  
        result :: add\_fields (in\_dml, validation\_fields);   
end ]   
  
  
Let’s write the transform — we want the transform to look like this:   
out :: reformat (in) =   
begin   
        out.\* :: in.\*;   
          out.first\_name\_is\_valid :: is\_valid(in.first\_name);   
          out.last\_name\_is\_valid :: is\_valid(in.last\_name);   
          out.address\_is\_valid :: is\_valid(in.address);   
end   
  
What steps do we need to do in metaprogramming to create this transform?   
1. Loop over the input fields and create an is\_valid output rule for each input field   
2. Add those rules to a simple reformat function that contains a wildcard rule.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let rules\_vec = for (let field in in\_fields) : make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                        “is\_valid(in.” + field + “)”);   
        let simple\_xfr = add\_rules (make\_transform (“reformat”, [vector make\_arg(“in”)]), “out.\*”, “in.\*”);   
        result :: add\_rules (simple\_xfr, rules\_vec);   
end]   
  
  
Question is:   
        can we combine these two block expressions into a single block expression that computes both the record format AND the transform?   
        that way, we need to loop over the in\_fields only once.   
  
        yes!   
  
        in my original code, I used for loop expressions   
                for loop expression works well when you are generating one vector   
  
        I want to use one loop to generate \*two\* vectors — one vector containing fields for   
                the output record & the other vector containing the rules for the transform   
        I need to use a loop statement to do that   
        In a loop statement, you use vector\_append to add new elements to an existing vector.   
  
$[ begin   
        let in\_fields = record\_info\_item(in\_dml, “name”);   
        let dml\_field\_info\_vec validation\_fields = allocate(); // must explicitly declare the type here   
        let dml\_rules\_vec rules\_vec = allocate(); // must explicitly declare the type here as well   
        let simple\_xfr = add\_rules (make\_transform (“reformat”, [vector make\_arg(“in”)]),   
                                                                        “out.\*”, “in.\*”);   
  
        for (let field in in\_fields) begin   
                validation\_fields = vector\_append (validation\_fields,   
                                                                                                make\_field(field + “\_is\_valid”, “integer(1)”));   
                rules\_vec = vector\_append (rules\_vec,   
                                                                                make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”));   
        end   
          
        result :: [record dml add\_fields (in\_dml, validation\_fields)     
                              xfr   add\_rules (simple\_xfr, rules\_vec) ];   
end ]   
  
Does anyone know how to return two (or more) values from a block expression or a function?   
I want this block expression to return \*BOTH\* at the same time, a record format and a transform.   
I can’t use if-else — that only returns 1 thing.   
What kind of data type can I used to store two or more values that have different data types?   
Vector stores a list of values that all have the same data type.   
Record stores a set of fields, each fields can have a different type!   
  
Let’s say that the block expression above is assigned to a parameter named “validation”   
So in my components, when I want to refer to the record format that is computed for the validation parameter, I write:   
        $validation.dml   
  
and when I want to refer to the transform that is computed for the validation parameter, I write:   
        $validation.xfr   
  
In a PDL expression   
        $[ validation.dml ]   
        $[ validation.xfr ]   
  
  
Option #1   
In the expression above, I used vector\_append and make\_rule/make\_field to create a vector of rules or a vector of fields. Then I used add\_fields/add\_rules at the end to create record format or transform.   
        for (let field in in\_fields) begin   
                validation\_fields = vector\_append (validation\_fields,   
                                                                                                make\_field(field + “\_is\_valid”, “integer(1)”));   
                rules\_vec = vector\_append (rules\_vec,   
                                                                                make\_rule (“out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”));   
        end   
          
        result :: [record dml add\_fields (in\_dml, validation\_fields)     
                              xfr   add\_rules (simple\_xfr, rules\_vec) ];   
  
  
Option #2   
You can use add\_field or add\_rule inside the loop.   
        let string(int) out\_dml = in\_dml;   
        let string(int) out\_xfr = simple\_xfr;   
        for (let field in in\_fields) begin   
                out\_dml = add\_field(out\_dml, field + “\_is\_valid”, “integer(1)”);   
                out\_xfr = add\_rule(out\_xfr, “out.” + field + “\_is\_valid”,   
                                                                                                       “is\_valid(in.” + field + “)”);   
        end   
          
        result :: [record dml out\_dml   
                              xfr   out\_xfr ];   
  
  
Option #2 is \*significantly\* slower when you have a large number of fields. Page #129 in the PPT slides shows the difference in performance.   
Option #1 is the preferred option:   
inside the for loop   
        create a vector of fields using vector\_append and make\_field   
        create a vector of rules using vector\_append and make rule   
  
at the end, add\_rules (xfr, rules\_vec)   
        and add\_fields (dml, fields\_vec)   
  
  
  
==========================================   
Workflow for building & testing a reusable component (linked subgraph)   
  
1- Build the graph with hardcoded inputs & outputs, and test it to make sure that it works.   
2- Parameterize the inputs & outputs   
                inputs are typically input parameters, values supplied by the end user   
                outputs are typically computed with metaprogramming   
Test it with various combinations of inputs to verify that it is truly generic and works in all cases.   
3- Take the components that you want to reuse, put them in a subgraph, and move the appropriate parameters to the properties of the subgraph. Now the parameters that the components in the subgraph uses are properties of that subgraph.   
Test again to make sure it still works   
4- Save the subgraph in a separate file (Select subgraph, then File > Save Subgraph … As). Then you’ve got a parameterized subgraph that can be reused (as a component) in other graphs.   
  
In Exercise #3 (the one with the Read Multiple Files), we’ve already done steps #1 and #2. The directory & the input record format are parameterized, and the output record format is computed using metaprogramming.   
I’m going to demonstrate how to do steps #3 and #4.   
  
  
If you need to make changes to the component that you’ve saved   
—> You right click the subgraph in the Sandbox View (in the components folder inside your sandbox) and choose “Edit”. Making changes to the subgraph itself.   
—> For any instances of that subgraph that are used in other graphs, you need to update the instances with those changes.   
        option #1 — open the graph, right-click the linked subgraph and choose Update   
        option #2 — “air sandbox update” — you can update multiple graphs & components at the   
                        same time   
  
==========   
  
What is the difference between PDL and metaprogramming?   
  
PDL is a parameter evaluation syntax   
        superset of constant, $-substitution, ${ } substitution, and shell interpretation   
        plus it has a “inline DML computation” syntax $[ ]   
  
        inside $[ ], you can write Ab Initio DML expressions and have them evaluated by the   
        Co>Operating System DML engine.   
  
        there are things that you can use PDL for that don’t involve metaprogramming   
        e.g., simple computations like today’s date as part of the file path.   
  
metaprogramming — design technique for generic graphs   
        typical design pattern for generic graph :   
                end user supplies input information (data set information + record formats) plus   
                some amount of information about what they want to do (keys, functions to use,   
                which fields to drop, etc.)   
  
                rest of the metadata — record formats, transforms, key — are computed with   
                metaprogramming functions   
  
        set of functions that you can use to compute record formats, transforms, and keys for   
        the graph to use   
  
        you can do metaprogramming without PDL — way back before PDL became available in   
                V2.14, you could have two graphs: first graph computes the transforms and record   
                formats + second graph uses those transforms & record formats   
  
        PDL enables you to compute the record formats and transforms as parameters so that they   
        can be computed in the same graph that uses them.   
  
========   
  
[option #1] You don’t have to create the whole parameter value like this:   
$[ “record string(10) first\_name; string(10) last\_name; “ + new\_field\_type + “ “ + new\_field\_name + “;\nend ]   
  
[option #2 ]You could also do this:   
record   
        string(10) first\_name;   
        string(10) last\_name;   
        $[ new\_field\_type + “ “ + new\_field\_name + “;”]   
end   
  
[option #3] And of course, you could also do this:   
$[ add\_field (“record string(10) first\_name; string(10) last\_name; end“, new\_field\_type, new\_field\_name) ]   
  
Always lots of ways to do things — think about the complexity.   
The more complex it is — the more likely it is better that you use metaprogramming functions like add\_field, add\_rule, etc.   
The simpler it is — the more likely it is better to just use simple string functions or to embed the computation inside a hardcoded string, like option #2 above   
  
  
======================================   
  
Is AB\_DML\_DEFS specific to each graph?   
Yes. A graph can only see the AB\_DML\_DEFS that is defined in that graph.   
  
However, you can define a value for AB\_DML\_DEFS at the project level — in your projects parameters — then all the graphs in that project can see that definition of AB\_DML\_DEFS.   
  
It gets tricky if you want to define AB\_DML\_DEFS at the project level and then also for a specific graph. The graph value overrides the value of AB\_DML\_DEFS from the project level.   
Help Library has a lot of information on the interaction of this.   
  
========================================   
  
After class today, I’ll email instructions for a case study exercise — if you want to try that on your own. Exercise is to build a generic graph on your own from the beginning based on requirements given to you.   
If you do try the case study, let me know & I’ll be happy to take a look at your design OR give you advice if you get stuck at some point.   
  
Your accounts on the training server are active until February 28 2016.   
These materials (PPT slides on your desktop, your sandboxes, etc) are available at any time, the server is up 24/7 (except for routine maintenance for about 1 hour on Monday nights EST).   
  
Definitely encourage you to take a look at the Differencing graph example — very nice example of good design for a generic graph, uses AB\_DML\_DEFS, lots of metaprogramming, etc.   
        Help > Examples > Differencing Graph   
  
If you later think of any general questions that you have — I’m happy to help you:   
        rbuchheit@abinitio.com   
  
For customer projects, you should talk to support@abinitio.com — remember just let them know what customer this is for.

**CASE STUDY:**

****